SCMKV: A Lightweight Log-Structured Key-Value Store on SCM

Zhenjie Wang, Linpeng Huang(✉), and Yanmin Zhu

Department of Computer Science and Engineering,
Shanghai Jiao Tong University, Shanghai, China
{zhenjie.wang,lphuang,yzhu}@sjtu.edu.cn

Abstract. Storage Class Memories (SCMs) are promising technologies that would change the future of storage, with many attractive capabilities such as byte addressability, low latency and persistence. Existing key-value stores proposed for block devices use SCMs as block devices, which conceal the performance that SCMs provide. A few existing key-value stores for SCMs fail to provide consistency when hardware supports such as cache flush on power failure are unavailable. In this paper, we present a key-value store called SCMKV that provides consistency, performance and scalability. It takes advantage of characteristics of key-value workloads and leverages the log-structured technique for high throughput. In particular, we propose a static concurrent cache-friendly hash table to accelerate accesses to key-value objects, and maintain separate data logs and memory allocators for each worker thread for achieving high concurrency. To reduce write latency, it tries to reduce writes to SCMs and cache flushing instructions. Our experiments show that SCMKV achieves much higher throughput and has better scalability than state-of-the-art key-value stores.

Keywords: Storage Class Memory · Key-value store · Memory management · Log structure

1 Introduction

Emerging Storage Class Memory (SCM) technologies such as phase-change memory (PCM) [15], spin-torque transfer RAM (STT-RAM) [10] and resistive RAM (ReRAM) [11] have been gaining great attentions from both academia and industry. They have both DRAM-like and disk-like features, such as byte addressability, low latency and persistency. The most promising solution to integrating SCMs into current computer systems is to attach them directly to the memory bus along with traditional DRAM. Thus it is possible to access SCMs through regular load/store instructions. Such hybrid volatile/non-volatile memories offer an opportunity to build more effective and voluminous storage systems such as file systems and databases.

X. Shi et al. (Eds.): NPC 2017, LNCS 10578, pp. 1–12, 2017.
DOI: 10.1007/978-3-319-68210-5_1

Nowadays, key-value stores such as LevelDB [9], and Dynamo [6] have been very important applications in many Internet companies like Google and Amazon. Many data-intensive services provided by these companies rely on fast access to data. However, today's state-of-the-art key-value stores are optimized for block-based devices (e.g. disks and SSDs). These stores rely on file systems to persist data to devices. There exist file systems designed for SCM such as PMFS [7] and NOVA [17]. However, this will introduce some overheads caused by the software layer of file systems and lost the chances to improve the performance of key-value stores by directly accessing SCMs via load/store instructions.

Many researchers have focused on in-memory key-value caches. Memcached [8] and MICA [12] are popular key-value caches. They act as lookaside caches, keep a small part of workloads in DRAM for fast access, and rely on key-value stores to hold all data. Figure 1 shows different roles key-value caches and stores play in today's software stack. These cache systems can't act as key-value stores on SCMs because they are designed for different purpose and don't consider differences between SCMs and DRAM and data inconsistency due to system crashes.

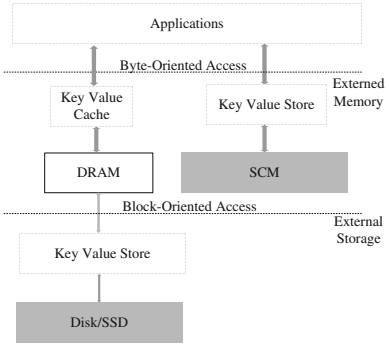

Fig. 1. Various key value systems on different storage devices

One of the most challenging problems for SCM key-value stores is consistency. Modern memory systems keep recently updated data on CPUs' caches and reorder stores to memory for performance. It may lead to inconsistency of data in the face of system crashes. To ensure key-value objects stored on SCM are consistent after recovery, writings of key-value objects must survive unexpected crashes. A simple method to persist data is to flush CPUs' caches explicitly and use memory barriers (e.g. sfence) to enforce write orders. However, frequently flushing CPUs' caches can introduce significant overheads and degrade the performance of the system. In this paper, we present a key-value store named SCMKV that takes advantage of characteristics of key-value workloads and adapts the log-structured technology to maximize throughput of SCMs while providing good scalability and consistency. SCMKV adapts conventional log-structured approach to manage memory, and appends new data to the sequentially written logs. It uses per-thread logs to avoid overheads caused by

synchronization primitives and lock-free data structures to provide high concurrency. It uses cache flushing instructions to persist metadata, and data corruption is detected by checksums when SCMKV recovers. Thus the number of cache flushing instructions is reduced greatly. Some information such as *next version number* and *current active log usages* can be constructed from the key-value store when SCMKV recovers. Thus they are allocated to DRAM and omitted from key-value store to reduce accesses to SCM.

The contributions of this paper are summarized as follows.

- It adapts existing log-structuring technology to SCMs and exploits the characteristics of key-value workloads to develop a key-value store.
- It designs a memory allocator for SCMs that moves some information to DRAM that reduces accesses to SCMs.
- It shows that SCMKV outperforms existing in-memory key-value cache and block-device optimized key-value store.

The remainder of the paper is organized as follows. Section 2 provides an overview of SCMKV Design. Section 3 gives details of SCMKV's implementation. Section 4 evaluates this work. Section 5 presents related work and Sect. 6 concludes.

2 Overview of SCMKV

SCMKV is a log-structured key-value store optimized for SCMs while taking advantage of the key-value workloads characteristics. Log-structuring technology is first introduced to build a file system to maximize disk bandwidth [14]. We adapt it to SCMs to offset the performance and wear out weaknesses of SCM and achieve more concurrency.

We designed SCMKV based on following observations. First, because the recently updated data are always appended to logs, the corruption of data due to power failure can only occur at the tail of logs. Thus it is easy to recover key-value store to the consistent state by scanning few logs. Second, there is only single log on storage systems based on disk because of the limited ability of disk sequential addressing. SCMs support random access, thus using multiple logs can achieve more concurrency. Third, a log of 2 MB can contain more than 4,000 key-value objects, if the size of each key-value object is less than 500 bytes. So we fix the size of the log and use small logs, thus it is easy to reserve contiguous free regions for garbage cleaning. Finally, according to [1], the sizes of key-value objects show strong modalities. Over 90% of SYS's keys are less than 40 bytes, and values sizes around 500 B take up nearly 80% of the entire cache's values. So given the size of an SCM device, we estimate the number of key-value objects it can contain.

Based on these observations, we have the following design decisions in SCMKV.

Maintaining Logs in SCM. SCMKV appends key-value objects to logs and inserts them into a hash table. Both logs and the hash table are located

in SCM. It doesn't cache updated key-value objects in DRAM as LevelDB. As we observed, the throughput of LevelDB for write-intensive workloads can be limited by the processing that merges data in DRAM to disk. It happens on SCM too. The logs' usages are often changed, thus they are kept in DRAM to reduce the latency of accesses to them. SCMKV can learn the logs' usages by scanning them when it recovers.

Not Persisting Key-Value Objects. Persisting data needs flushing CPUs' caches to SCM and restricting write orders. Both operations can increase the latency of writes. Furthermore, persisting key-value objects will need many flushing instructions, because their sizes are often bigger than the size of the cache line. Instead, SCMKV doesn't flush appended key-value objects. It only flushes some metadata of the hash table, and stores key-value object's checksums in SCM. When power fails, some key-value objects in the end of logs may be corruptible. SCMKV detects errors in active logs, discards them and rolls back to the latest valid key-value objects when it recovers. After SCMKV boots up, it is in the consistent status and doesn't need to check errors anymore.

Using Static Hash Table and Dynamic Chains. Many dynamic hashing schemes have been developed to index keys. These schemes allow the size of the hash table to grow and shrink gracefully according to the numbers of records. However, they cause some overheads when enlarging the size of the hash table. The size distribution of key-value objects can be predicted in the real scenario. Thus we can use a static hash table as the index of the store. To reduce the size of the statically allocated hash table, buckets of the hash table are 64-bit words. Each word contains a pointer point to a key-value object or a chain of the hash table.

3 Design and Implementation of SCMKV

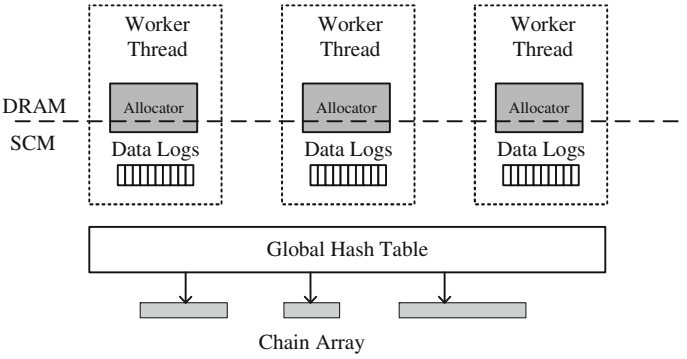

Fig. 2. The architecture of SCMKV.

3.1 Architecture

SCMKV is a persistent key-value store whose permanent data is maintained in non-volatile SCM. Figure 2 shows the architecture of SCMKV. It supports multiple threads to access key-value objects. To achieve good scalability, it uses per-thread allocators and per-thread data logs. These allocators are put on DRAM to support fast allocation. All threads share a global hash table to index key-value objects. The hash table uses dynamically allocated arrays to resolve hash collisions. Using arrays instead of lists can accelerate accesses to key-value objects by taking advantage of CPU's hardware prefetching. The source code is available on GitHub: https://github.com/page4/scmkv.

3.2 Memory Layout

Figure 3 shows the memory layout of SCMKV. The entire volume is divided into five segments. Superblock has the basic partition information and parameters of the store, which are not changeable after the store is created. Checkpoint keeps the status of the store, such as a pointer to free page list, locations of current active logs. A successful checkpoint gives a consistent status that enables the store to recover fast. The Page Information Table (PIT) contains per-page information. Each page contains a data log or some chains of the hash table. An element in the table contains information of the corresponding page, such as the number of key-value objects or chains, the size of live data and the time when the page is modified. For a free page, the element contains a pointer to next free page. The static hash table is a global hash table to locate key-value objects or its chains. Its size is not changed since the store is created. Data Area (DA) is a set of 2 MB pages. Each page is allocated and typed to be *data* or *chain*. A data page contains key-value objects, while a chain page contains chains of the hash table. A page does not store data and chains simultaneously.

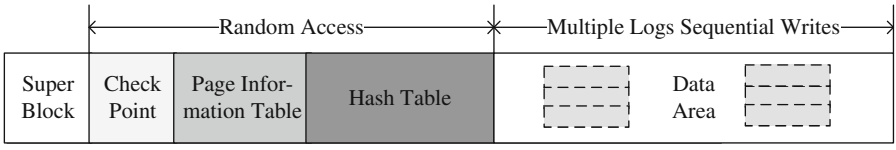

Fig. 3. SCM memory layout

3.3 Concurrent Cache-Aware Hash Table

Figure 4 shows our compact, concurrent and cache-aware hashing scheme. The numbers in the figure indicate the number of living key-value objects in a bucket, which may be smaller than the size of the chain. The bucket contains the address of a key-value object or an array. SCMKV accesses key-value objects directly when no collisions happen, and allocates dynamic arrays as chains when collisions occur. The implementation details are as follows.

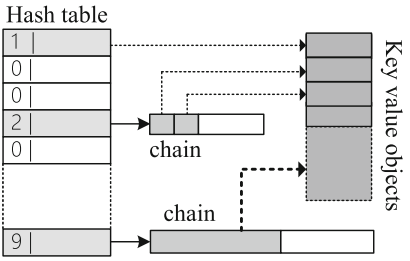

Fig. 4. Hash table structure

```
uint64_t htable_size;
struct hash_table_t {
  uint64_t has_writer:1;
  uint64_t reserved:7;
  uint64_t nr_items:8;
  uint64_t scm_addr:48;
} g_htable[htable_size];
```

Fig. 5. Layout of hash table

Buckets Management. Figure 5 shows the representation of the hash table in C/C++. It is used to give the details of Fig. 4. A bucket in the hash table is a 64-bit word. It allows multiple readers and a single writer to access the same bucket. The first field *has_writer* is used to guarantee only one writer is visiting the bucket. It is set or cleared by atomic operations. The third field *nr_items* indicates the number of key-value items located in the bucket. If *nr_items* is 1, *scm_addr* is the address of the key-value object in SCM. If *nr_items* is greater than 1, *scm_addr* is the address of a chain in SCM. Each chain contains the addresses of key-value objects.

Chains are cache aligned arrays that are dynamically allocated in the Data Area. They are multiple of cache line size. When a chain is full, SCMKV allocates a new chain whose size increases by a cache line size, copies data in the old chain to the new chain and sets *scm_addr* to the new chain.

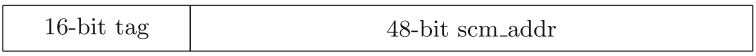

Fig. 6. Layout of a chian element

Cache-Friendly Lookup. An element in a chain is a 64-bit word as shown as Fig. 6. The 48-bit *scm_addr* is the address of a key-value object on SCM. The 16-bit *tag* field is a short summary of a key and can be represented as the high 16 bits of the key's hash. To keep the hash table compact, the actual keys are not stored in the hash table. When SCMKV lookups a key in a chain, it matches *tag*s, retrieves the inspected key and compares the retrieved key with the target key. *Tag* helps to avoid unnecessary retrieves and comparisons of keys. It is possible that two different keys have the same tag. However, with a 16-bit tag, the chance of a collision is only $1/2^{16} = 0.000015$. For a negative lookup operation that checks a bucket with 16 candidate elements, it makes $0.000015 * 16 = 0.00024$ pointer dereferences on average.

Consistency. SCMKV stores a key-value object (*kvobj*), the object's checksum (*kvobj_chsum*), the object's size (*obj_size*) and address of the old object with

the same key (*addr_of_oldobj*). It persists *obj_size* and *addr_of_oldobj* by flushing CPUs' cache. While the object is not guaranteed to be durable. If an object is corruptible, it can be rolled back to the old object by recursively visiting *addr_of_oldobj*.

3.4 Memory Allocator

In this section, we describe the per-thread memory allocators on SCM. Each thread in SCMKV has its own memory allocators, which reduces overheads caused by synchronization primitives. SCMKV uses a multilevel memory allocation model to manage memory effectively. At the bottom of this model is the page management layer which keeps a pool of empty pages, allocates and releases pages. At the middle of this model are log allocators. The allocator requires a page from the bottom level when a new log is allocated. It will release the page to the bottom level if a page becomes empty. At the top are object allocators which allocate memory for chains or key-value objects. These object allocators require memory from the tail of logs. They can't reuse space before the tail of logs. Thus there exist many unavailable holes in the logs when key-value objects are deleted. To reuse these holes, allocators need to perform garbage collection. Garbage collection moves some living key-value objects in the evicted log to a new log, and release the page of the evicted log to the bottom level (Fig. 7).

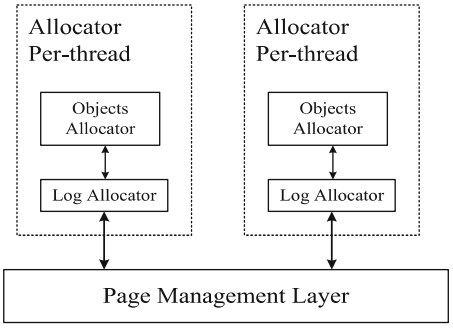

Fig. 7. SCM memory allocators

Keep Current Page Information in DRAM. For every page, there is an element in the PIT that records usages of the page. Page usages are often changed for write-intensive workloads. To reduce the number of writes to SCM, log allocators copy the element to DRAM, thus updates of page usages are kept in DRAM. When the page becomes full, the log allocator stores its usages to SCM, persists data in the log using *clwb* or *clflushopt* instructions and requires a new page from the memory pool to start a new log.

Lock-Free Updates when Allocating Memory. Due to per-thread logs, there are no multiple threads appending key-value objects to the same log. However, page usages can be modified by two threads when they want to remove key-value objects in the same log. So the updates of page usages must be atomic. SCMKV uses atomic writes or compare-and-swap (CAS) to guarantee consistency.

3.5 Garbage Collection

SCMKV uses a *Garbage Collection* thread to reclaim free memories that accumulate in the logs when key-value objects or chain arrays are deleted. After SCMKV boots up, the thread scans the PIT to learn usages of the pages, then it uses a similar cost-benefit approach as LFS [14] to select evicted logs. The evicted logs are chosen based on the amount of free space in the log and the age of the log. The age of log is estimated by the time when the log is modified. For each of the chosen logs, the garbage collection thread scans key-value objects stored in the log, copies live objects to a new log and re-insert them to the store. Then it releases the evicted logs to the memory pool, making the evicted logs' memory available for new logs.

3.6 Shutdown and Recovery

When SCMKV shuts down normally, it flushes all current active logs and some data structures (e.g. allocators) to SCM. Thus it can recover fast. In the case of unclean shutdown (e.g., system crash), SCMKV must recover to the consistent status and rebuild some data structures by scanning the data logs. The recovery process is fast due to following designs. First, the size of logs is not bigger than the size of a page. Second, when a log is full, key value objects in the log are persisted to SCM by memory allocators. Thus there are only several possible unclean logs. Third, SCMKV starts a recovery thread for each possible unclean log. The recovery thread scans all key-value objects in its log, records the first corruption object, sets the tail of the log to the address of the object, and pushes the remaining objects' *addr_of_oldobj* to a recovery stack. In the last, it rolls back all the objects in the stack.

4 Evaluation

We now describe the experimental setup for the evaluation of SCMKV. Then we present results from a detailed evaluation with several micro-benchmarks.

Experimental Setup. In our test, we use a Dell-R730 server running Linux 2.6.32. It is equipped with dual 10-core CPUs (Intel Xeon E5-2650-V3 @2.3 GHz, Ivy Bridge EP). Each CPU has 25 MB of L3 cache and each core has 10×256 KB of L2 cache. The total size of DRAM is 128 GB. We use DRAM to emulate SCM, thus some features of SCMKV can't be evaluated.

Table 1. Datasets with different key value size

Dataset	Key size	Value size	Count
Small	16 B	100 B	64 Mi
Large	128 B	512 B	8 Mi

In the evaluation, we compare SCMKV with LevelDB [9] and Masstree [13]. LevelDB relies on file systems to persist data. To test LevelDB, we reserve a continuous memory area from OS when Linux boots. An ext4 file system is created on the reserved memory. Masstree is a typical in-memory key-value store, and it has outperformed other stores as reported by [12,13]. In our evaluation, data of all these systems are stored in memory, and it will not involve any disk or network overheads.

Benchmark. We use Yahoo's YCSB [4] benchmark suite to generate read and write requests. The datasets are shown in Table 1. These datasets are generated according to two workload types: *uniform* and *skewed*. Uniform workload generates an item uniformly at random, while skewed workload generates an item according to Zipfian distribution.

4.1 Write/Read Throughput of Single Thread

Figure 8 plots the write throughput of a single thread. To show the influences of enforcing write orders, SCMKV has been run in two modes: in *scmkv-noflush* mode, it doesn't use any flush or memory barries instructions, while in *scmkv-flush* it uses *clflush* and *sfence* to enforce write orders. We don't enforce write orders in Masstree and LevelDB. The overall performance of scmkv-noflush is

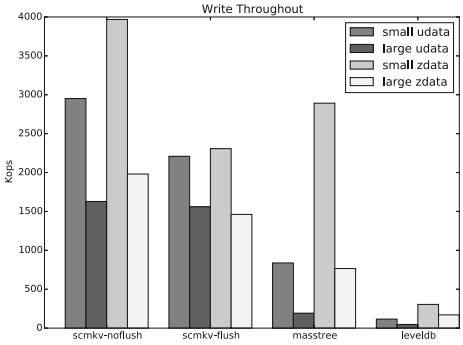

Fig. 8. Single Thread Write Throughput of Different Datasets. *udata indicates uniform workloads, and zdata refers to skewed (Zipfian distribution) workloads.*

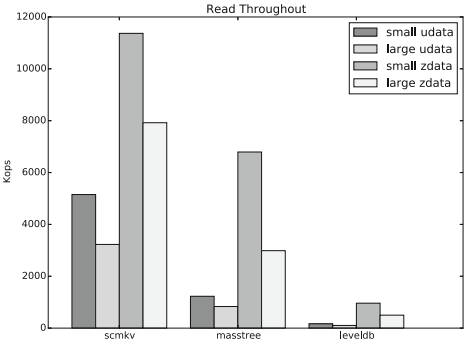

Fig. 9. Single Thread Read Throughput of Different Datasets. *udata indicates uniform workloads, and zdata refers to skewed (Zipfian distribution) workloads.*

better than Masstree. With enforcing write orders, SCMKV suffers 40% performance reducing for small key-value objects and 20% performance reducing for large key-value objects in the skewed workloads. As expected, SCMKV has lower overheads caused by cache flushing on larger key-value objects.

In the following descriptions, we compare scmkv-flush with Masstree and LevelDB. As shown in the Fig. 8, SCMKV has 2.6x throughput in small uniform datasets and 7x throughput in large uniform datasets than Masstree. And it has 10x throughput in small uniform datasets and 30x throughput in larger uniform than LevelDB. There are similar results in large skewed datasets. However, SCMKV has smaller throughput than Masstree in the case of small skewed datasets. This is limited by overheads caused by write orders. Besides, the throughputs of skewed datasets are almost same as uniform data in scmkv-flush because SCMKV doesn't benefit from the data locality of skewed datasets. We expect the occurrence of instruction *CLWB* will improve this problem in future.

Figure 9 plots the read throughput of a single thread. Like Masstree, SCMKV returns the pointer to the *value* and the size of value when retrieving the value of a *key*. SCMKV performs better than Masstree and LevelDB in both uniform and skewed datasets due to the efficiency of our design choices. Throughputs of skewed datasets are better than uniform datasets because of the data locality of workloads.

4.2 Scalability

Figure 10 shows write throughputs of small uniform datasets with the increase in number of cores. Both SCMKV and Masstree scale well, while the throughput of LevelDB grows from 0.11Mops to 0.21Mops when we augment the number of cores from 1 core to 10 cores. SCMKV achieves 2.1Mops at 1 core and 5.4Mops at 4 cores. Due to frequency of hash collisions raising with number of cores, throughput per core becomes decreasing. SCMKV has a throughput of 6.7Mops at 10 cores.

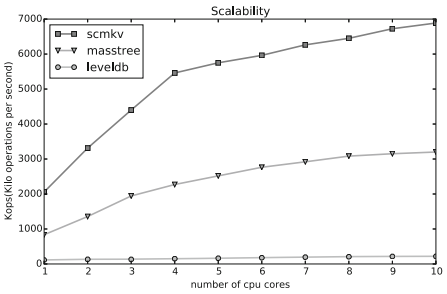

Fig. 10. Write throughput of small uniform dataset using a varying number of cores.

5 Related Work

In this section, we discuss previous studies on storage systems related to SCM and key-value systems.

Storage Class Memory Systems. Many efforts have been devoted to providing different abstractions on SCM. NV-heaps [3] and NVML [5] provide general programming interface with persistent memory and transaction mechanisms for failure recovery. NOVA [17] and PMFS [7] are file systems optimized for persistent memory. They allow for traditional file-based access to memory by POSIX file interface. PMFS uses multi-granularity atomic updates with different CPU instructions and fine-grained logging for metadata consistency and Copy on Write for data consistency.

Key-Value Systems. Key-value systems have been always been optimized for storage media. There are many systems optimized for disk, flash and DRAM. For example, LevelDB is a system based on log-structured merge trees to reduce the latency of disk by sequential writes to disk. NMVKV is a flash-aware key-value store and relies on the Flash Translation Layer (FTL) capabilities to minimal data management at the key-value store.

There are also some key-value systems optimized for Non-volatile Memory. NVMcached [16] is a key-value cache for the non-volatile memory that tries to avoid most cache flushes by using checksums to weed out corrupted data. It acts as a lookaside cache and requires re-inserted for any lost key-value objects. Echo [2] presents a persist key-value storage system that using two-level memory architecture that combining DRAM and SCM. It employs snapshot isolation to support concurrency and consistency, keeps a set of local stores on DRAM for each *worker threads* and a master store on SCM for *master threads*.

6 Conclusions

This paper presents a key-value store optimized for Storage Class Memory (SCM). It is intended to effectively manage storage class memory. It uses per-thread data structures to reduce competitions among threads. And it also reduces the number of some expensive operations such as cache flushing (e.g. unnecessary writes to SCM, explicitly cache flushing) to lower write latency. Experimental results have shown that it achieves high throughput and good scalability for both uniform and skewed datasets.

Acknowledgment. This work is supported in part by National Natural Science Foundation of China (No. 61472241, 61472254, and 61170238) and the National High Technology Research and Development Program of China (No. 2015AA015303). This work is also supported by the Program for New Century Excellent Talents in University of China, the Program for Changjiang Young Scholars in University of China, and the Program for Shanghai Top Young Talents.

References

1. Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., Paleczny, M.: Workload analysis of a large-scale key-value store. In: ACM SIGMETRICS Performance Evaluation Review, vol. 40, pp. 53–64. ACM (2012)
2. Bailey, K.A., Hornyack, P., Ceze, L., Gribble, S.D., Levy, H.M.: Exploring storage class memory with key value stores. In: Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads, p. 4. ACM (2013)
3. Coburn, J., Caulfield, A.M., Akel, A., Grupp, L.M., Gupta, R.K., Jhala, R., Swanson, S.: Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. ACM Sigplan Not. **46**(3), 105–118 (2011)
4. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 143–154. ACM (2010)
5. Corporation, I.: Nvm library (2017). https://github.com/pmem/nvml
6. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. ACM SIGOPS Operating Syst. Rev. **41**(6), 205–220 (2007)
7. Dulloor, S.R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., Jackson, J.: System software for persistent memory. In: Proceedings of the Ninth European Conference on Computer Systems, p. 15. ACM (2014)
8. Fitzpatrick, B.: Distributed caching with hmemcached. Linux J. **2004**(124), 5 (2004)
9. Ghemawat, S., Dean, J.: Leveldb (2011). https://github.com/google/leveldb
10. Hosomi, M., Yamagishi, H., Yamamoto, T., Bessho, K., Higo, Y., Yamane, K., Yamada, H., Shoji, M., Hachino, H., Fukumoto, C., et al.: A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-ram. In: IEEE International Electron Devices Meeting, IEDM Technical Digest, pp. 459–462. IEEE (2005)
11. Kawahara, A., Azuma, R., Ikeda, Y., Kawai, K., Katoh, Y., Hayakawa, Y., Tsuji, K., Yoneda, S., Himeno, A., Shimakawa, K., et al.: An 8 mb multi-layered cross-point reram macro with 443 mb/s write throughput. IEEE J. Solid-State Circ. **48**(1), 178–185 (2013)
12. Lim, H., Han, D., Andersen, D.G., Kaminsky, M.: Mica: A holistic approach to fast in-memory key-value storage. Management **15**(32), 36 (2014)
13. Mao, Y., Kohler, E., Morris, R.T.: Cache craftiness for fast multicore key-value storage. In: Proceedings of the 7th ACM European Conference on Computer Systems, pp. 183–196. ACM (2012)
14. Rosenblum, M., Ousterhout, J.K.: The design and implementation of a log-structured file system. ACM Trans. Comput. Syst. (TOCS) **10**(1), 26–52 (1992)
15. Wong, H.S.P., Raoux, S., Kim, S., Liang, J., Reifenberg, J.P., Rajendran, B., Asheghi, M., Goodson, K.E.: Phase change memory. Proc. IEEE **98**(12), 2201–2227 (2010)
16. Wu, X., Ni, F., Zhang, L., Wang, Y., Ren, Y., Hack, M., Shao, Z., Jiang, S.: Nvm-cached: An nvm-based key-value cache. In: Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems, p. 18. ACM (2016)
17. Xu, J., Swanson, S.: Nova: a log-structured file system for hybrid volatile/non-volatile main memories. In: FAST, pp. 323–338 (2016)